Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática

INF 2060: Estudo Orientado

MoCA Proxy Framework

Manual de utilização

Hana Karina Rubinsztejn

Prof. Markus Endler

Índice

1 MoCA ProxyFramework

A arquitetura MoCA (Mobile Collaboration Architecture) é uma arquitetura de middleware para aplicações colaborativas para usuários móveis. A arquitetura MoCA oferece um conjunto de serviços que coletam e distribuem informações do contexto de execução de cada dispositivo. Aplicações baseadas na MoCA são aplicações compostas por três partes: servidor, proxy e cliente (móvel). O proxy na MoCA tem o papel de intermediar a comunicação entre servidor e clientes, executando adaptações, se necessário, de acordo com o estado do cliente.

O proxy framework tem o objetivo de facilitar o desenvolvimento da aplicação proxy, sendo responsável por interagir com a MoCA, escondendo detalhes desta interação e inscrevendo-se como interessado por notificações sobre os contextos de interesse da aplicação, para cada cliente cadastrado pela aplicação. O proxy framework oferece facilidades para a configuração de contextos de interesse, e associação das ações que devem ser tomadas quando da ocorrência de tais contextos (ou estados). A Figura 1 mostra um esquema geral da atuação do Proxy.

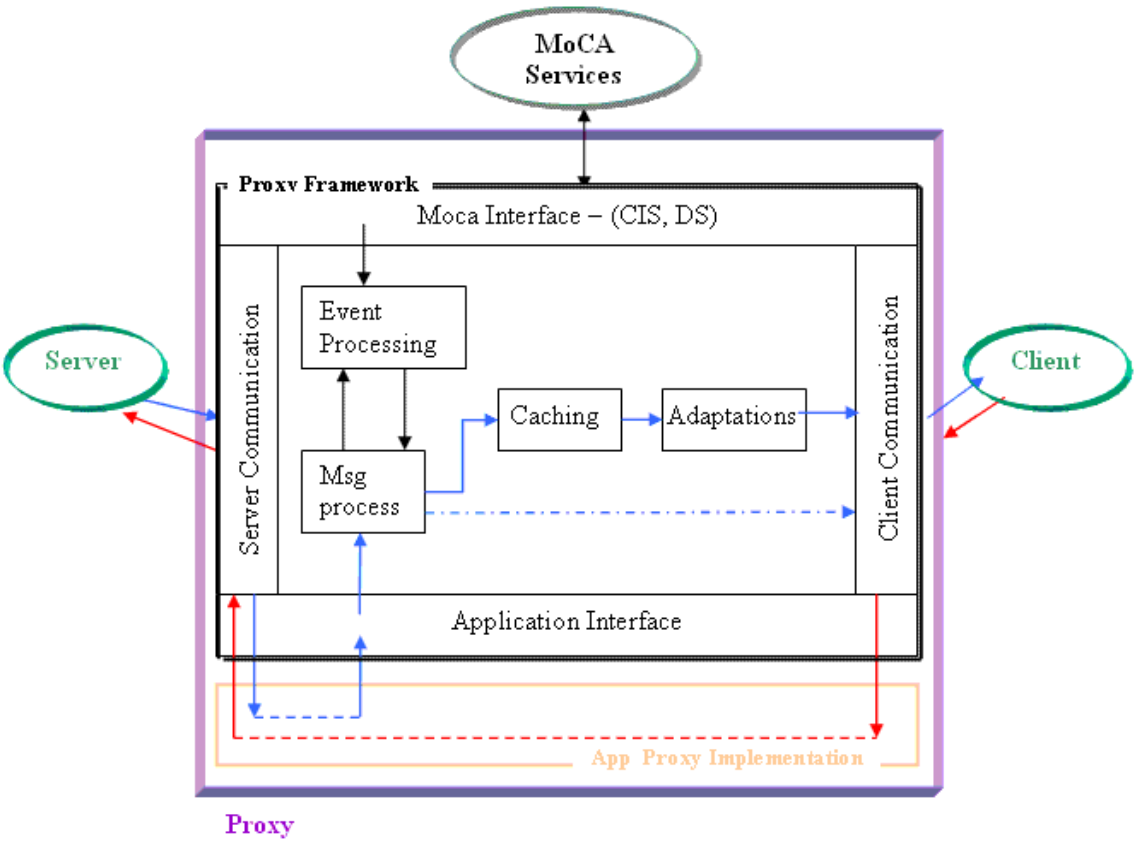

Figura 1: Visão geral do Proxy.

O Framework é composto por componentes de comunicação com clientes e servidor, e pelo núcleo responsável pela comunicação com o CIS e pelo mecanismo de aplicação de adaptações, além de uma interface de interação entre o framework e a implementação proxy.

A interface ProxyFacade, do pacote `proxy.development`, apresenta todos os métodos para a interação com o framework. Para utilizar as funcionalidades do framework deve-se utilizar a classe ProxyFramework para acessar as funções disponíveis para a interação entre as classes da aplicação e o framework.

A classe ProxyFramework oferece métodos para inicialização e configuração do proxy, gerência de usuários, envio e recebimento de mensagens para clientes e servidor, recebimento de notificações de conectividade e liberação de recursos.

Deve-se notar que para usar o framework, deve-se obter a instância da classe ProxyFramework e em seguida configurar o endereço da aplicação servidora. Para iniciar o proxy, deve-se utilizar o método **`init`**. Este método deve ser o primeiro método a ser invocado, pois além da comunicação com o servidor, ele cria servidores de comunicação com clientes e configura as regras que definem as adaptações a serem aplicadas de acordo com o contexto do cliente. Esta configuração é feita a partir do arquivo ProxyConfig.xml, cujo formato é definido na seção 4.

Deste ponto em diante, considere usuário do framework como sendo o desenvolvedor da aplicação que instancia o framework e utiliza suas funcionalidades, e considere cliente como o cliente móvel da aplicação. Além disso, aplicação proxy é a parte específica do proxy desenvolvida pelo usuário do proxyFramework.

2 Comunicação com servidor e clientes

O proxy intermedeia toda a comunicação entre clientes móveis e o servidor da aplicação. Para a troca de mensagens, ele utiliza de uma interface comum definida em `proxy.message.Message`. O framework disponibiliza uma implementação padrão para as mensagens em `proxy.message.DefaultMessage`, que deve ser utilizada pelo desenvolvedor. Esta classe pode ser estendida se desejar. Um detalhe é que a classe deve ser clonável, portanto deve-se na nova classe sobrescrever o método clone se necessário.

Toda mensagem recebida pelo framework é repassada para a aplicação proxy, que deve decidir se a mensagem deve ou não ser retransmitida. Isso é implementado através de callbacks. Assim, no desenvolvimento da aplicação Proxy, deve-se implementar uma interface listener `MessageListener` para o recebimento das mensagens (implementar `onReceiveMessage`). Esta interface é encontrada no pacote `proxy.development`.

Para que a aplicação proxy receba mensagens vindas dos clientes deve-se cadastrar no framework um `MessageListener` através do método `addClientMsgListener` da classe `ProxyFramework`. Para recebimento de mensagens vindas do servidor deve-se cadastrar um `MessageListener` através do método `addServerMsgListener`.

O framework permite dois tipos de comunicação, síncrono e assíncrono, com ambas as partes. A aplicação Proxy é responsável por definir o tipo de comunicação a ser utilizada e pela decisão de quais mensagens devem ser retransmitidas. O framework é responsável por realizar conversões de protocolos, se necessário, para o envio de mensagens aos clientes móveis. A utilização destes dois tipos de comunicação é explicada a seguir.

Observação: Todos os métodos utilizados na troca de mensagens, mostrados a diante, são definidos na classe `proxy.ProxyFramework`.

Comunicação Síncrona

A comunicação síncrona entre a aplicação proxy e os clientes ou servidor é realizada através dos métodos `sendMsgToClient e sendMsgToServer`, respectivamente. A aplicação deve apenas informar o destinatário, e o framework se encarrega de entregar a mensagem utilizando o protocolo de comunicação adequado ao cliente.

Comunicação Assíncrona

A comunicação assíncrona publish/subscribe do framework é baseada no serviço de eventos da MoCA[1], com algumas alterações. Neste tipo de comunicação o framework atua

[1] Maiores informações consulte www.lac.infpuc-rio.br/moca/ECI

como um servidor de eventos na comunicação com os clientes móveis e como “subscriber” na comunicação com o servidor da aplicação.

Permite-se apenas publicação por tópicos e a aplicação proxy é responsável por solicitar ao framework para se inscrever ou desinscrever (“subscribe/unsubscribe”) no servidor da aplicação para cada tópico de interesse. Além disso, ela é também responsável por publicar no servidor mensagens vindas de um cliente para um tópico específico, e vice-versa.

Para que a aplicação proxy receba notificações de pedidos de subscrições vindas dos clientes, é preciso implementar a interface `SubscribeListener` encontrada no pacote `proxy.development`. Implementando o método `onReceiveSubscription`, toda vez que um cliente é inscrito como interessado em um evento, ou quando ele se desregistrar, a aplicação Proxy será informada. Para isso, é necessário também cadastrar no framework este `SubscribeListener` através do método `addSubscriptionListener` da classe `proxy.ProxyFramework`.

Estas notificações podem ser usadas pela aplicação Proxy, para manter uma lista de interessados para cada tópico, e decidir quando subscrever em um tópico no servidor, e quando se desinscrever. Por exemplo, na primeira subscrição por um tópico, a aplicação Proxy pode se subscrever no servidor. Se outros clientes tiverem interesse pelo mesmo tópico, não é necessário fazer este pedido de novo ao servidor. Quando o último cliente interessado em um tópico se desregistrar (unsubscribe), a aplicação Proxy também pode desinscrever este tópico no servidor.

Para o registro de interesse em um tópico no servidor, a aplicação Proxy deve utilizar o método `subscribeToServer`. Para o cancelar o interesse use `unsubscribeFromServer`.

A publicação de mensagens de um tópico para o servidor ou para os clientes interessados deve ser feita utilizando os métodos `publishMsgToServer`, `publishMsgToClients`, respectivamente. É importante salientar que a publicação para clientes também é feita para um tópico (e não clientes específicos), e o framework é responsável por identificar quais os clientes interessados no tópico e enviar a mensagem a eles, efetuando adaptações caso necessário.

Mensagens postadas por clientes para um tópico são recebidas pela aplicação Proxy através do método `onReceiveMessage da interface MessageListener`, como explicado anteriormente. Se a mensagem for de um tópico cadastrado no servidor, é função da aplicação proxy publicar esta mensagem ao servidor e demais clientes, se desejar.

3 Extensão de ações

O ProxyFramework permite que determinadas ações sejam executadas de acordo com o estado corrente do cliente. Por serem específicas para cada aplicação, as ações devem ser implementadas pelo desenvolvedor do proxy, que deve criar seus próprios mecanismos de adaptação.

As ações são definidas pela classe base Action do pacote `proxy.actions`. Esta classe fornece métodos comuns, como por exemplo, o de recuperação de parâmetros (`getParameter`). Existem dois tipos de ações: as de adaptação de mensagens (chamadas *adapters*) e as de mudança de estado (chamadas *listeners*).

As ações do tipo adapter são ações acionadas no momento do envio de uma mensagem a um cliente, dependendo de seu contexto corrente. Para criar uma ação deste tipo, deve-se estender a classe abstrata proxy.actions.adpter.Adapter. Esta classe possui o método `execute`, que recebe informações sobre o destinatário da mensagem e a própria mensagem a ser adaptada. O retorno deste método é a mensagem processada pelo adapter, ou seja, a mensagem alterada através da aplicação da estratégia implementada. Note que se o retorno de um adapter for Null o fluxo de adaptações é interrompido, já que a mensagem foi descartada.

As ações do tipo *listeners* reagem às mudanças no estado de clientes. Para implementar um listener, basta estender a classe abstrata `StateListener` do pacote `proxy.actions.listener`. Esta classe possui dois métodos: `matches` e `unmatches`. O primeiro é executado sempre que o estado muda de OFF para ON, e o segundo de ON para OFF. Em ambos os casos, são fornecidas as informações do cliente que sofreu a mudança de estado e uma referência para o dispatcher, permitindo que o listener possa mandar mensagens para outros receivers.

Uma observação: caso a ação criada possua parâmetros configuráveis no XML, estes não podem ser recuperados no construtor da classe estendida (apenas depois do método `init` de Action). Outro detalhe é que as instâncias das ações são reaproveitadas em todas as execuções dentro do proxy. Assim, recomenda-se que as implementações não armazenem nenhum tipo de estado através de variáveis de instância.

4 Configuração de contextos e ações

Os estados (ou contextos) a serem monitorados, bem como as ações que devem ser aplicadas em cada estado, devem ser definidos no arquivo XML, ProxyConfig.xml.

Neste arquivo, cada estado é definido por uma expressão de interesse. Cada estado pode possuir uma ação do tipo listener, e um número não determinado de regras para adaptação de mensagens. Cada regra é composta por um filtro e ações de adaptação. O filtro serve pra validar as mensagens que podem ser adaptadas por determinado adaptador. As ações dentro da regra são aplicadas na ordem em que aparecerem no arquivo XML. Além disso, cada regra possui uma prioridade, que se não for definida será considerada como de menor prioridade. Se diferentes regras tiverem a mesma prioridade, serão aplicadas na mesma ordem em que foram definas.

Na Figura 2 é apresentado um exemplo de um arquivo de configuração. Neste exemplo, podemos ver o elemento State. Este elemento representa um estado monitorado. Todo elemento State possui um único elemento Expression. O elemento Expression corresponde à expressão[2] de monitoramento que será cadastrada no CIS da MoCA para verificação periódica da situação dos clientes.

No momento em que o estado de um cliente transiciona de ON para OFF ou vice-versa, o usuário do framework pode customizar ações *listeners* a ser executadas. Estas ações serão implementadas estendendo-se a classe StateListener e sua configuração será através do elemento Action.

Cada estado pode ter um número ilimitado de elementos Rule. Os elementos Rule correspondem as regras. Regras são agrupamentos de adaptadores que serão aplicados caso o estado para o qual foram cadastradas esteja ON e uma determinada condição da mensagem e/ou do seu destinatário seja satisfeita. A condição é determinada através do elemento Filter. O usuário pode configurar um filtro utilizando-se de uma série de operadores e elementos disponíveis, que são mais bem explicados na próxima sub-seção.

Uma vez que o filtro tenha aceitado a mensagem, a série de adapters cadastrados para a regra será executada.

Adaptadores também devem ser cadastrados à uma regra usando-se o elemento Action, com a única diferença de que a classe cadastrada deve estender Adapter.

É possível fornecer parâmetros tanto para os *listeners* quanto para os *adapters*. Estes parâmetros são passados utilizando-se o elemento Parameter (veja exemplo). Cada parâmetro possui um nome e um valor. Os parâmetros são exclusivos de cada classe, e seus valores podem ser recuperados através do nome, ou seja, o nome é utilizado como chave e por isso ele é case sensitive.

---

[2] Para a definição das expressões de contexto consulte www.lac.inf.puc-rio.br/moca/CIS

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ProxyConf>
  <State>
      <Expression>
          <![CDATA[ OnLine = false AND DeltaT > 3000 ]]>
      </Expression>
      <Action class="moca.core.proxy.actions.listeners.DefaultCacheListener">
         <Parameter name="cacheClassName">moca.core.proxy.cache.FIFOCacher
         </Parameter>
      </Action>
  </State>
  <State>
      <Expression>
          <![CDATA[ CPU > 60  AND FreeMemory < 10000 ]]>
      </Expression>
      <Rule priority="1">
          <Filter>
              <And>
                 <!-- tipo de dados da mensagem -->
                 <StartWith>
                    <FieldValue>
                      <Literal>datatype</Literal>
                    </FieldValue>
                    <Literal>image/</Literal>
                 </StartWith>
                 <!-- e protocolo  -->
                 <Equals>
                    <FieldValue>
                      <Literal>protocol</Literal>
                    </FieldValue>
                    <Literal>TCP</Literal>
                  </Equals>
              </And>
          </Filter>
          <Action class="moca.core.proxy.actions.adapters.ConvertToJPEGAdapter">
              <Parameter name="compressionQuality">0.3</Parameter>
          </Action>
      </Rule>
  </State>
</ProxyConf>
```

Figura 2: Exemplo de arquivo de configuração

Observação 1: É necessária a utilização de uma seção CDATA na especificação da expressão de contexto, para indicar ao XML *parser* que a região pode conter qualquer string. Ex:

<![CDATA[CPU > 60 AND FreeMemory < 10000]]>]

Isto é necessário porque a expressão de contexto contém caracteres que fazem parte da sintaxe XML, como por exemplo, ‘<’ e ‘>’.

Observação 2: Na denominação dos parâmetros (elemento Parameter) das ações, as diferenças entre letras maiúsculas e minúsculas são levadas em consideração (case sensitive).

Observação 3: As regras (Rule) só podem possuir ações do tipo Adapter, o que significa também que não se pode aplicar filtros nas ações do tipo Listener. Outro detalhe é que não é possível a criação de uma mesma ação (listener) com parâmetros diferentes no mesmo estado. Entretanto, uma mesma ação adapter pode ser utilizada com parâmetros diferentes dentro de um mesmo estado, mas em regras diferentes.

Observação 4: Não há nenhuma restrição às condições do cliente se casarem com mais de um estado, ou seja, todas as ações de adaptação de todos os estados serão executadas (se passarem pelos filtros), e a ordem de execução dos estados será provavelmente a mesma do XML, mas não é garantida

4.1 Filtros

Os filtros são utilizados para selecionar as mensagens nas quais devem ser aplicadas as adaptações indicadas ao contexto corrente. Os tipos de elementos que podem ser filtrados dizem respeito à mensagem (e seu tipo de conteúdo) e/ou ao cliente a que se destina a mensagem.

Os elementos de filtragem disponíveis no momento são:

Elemento de Filtro	Descrição	Valores
communicationMode	Modo de comunicação do cliente	▪ SYNCHRONOUS: comunicação síncrona ▪ ASYNCHRONOUS: comunicação assíncrona
protocol	Protocolo de comunicação, utilizado pelo cliente	▪ TCP ▪ UDP

dataType	Tipo de dado da mensagem - em formato MIME, em geral	▪ Ex: "image/jpeg"
Client	identificador do destinatário	
Subject	Assunto da notificação a ser publicada	

A especificação de um elemento de filtragem no arquivo XML é feita através das tags **FieldValue** e **Literal**. FieldValue indica que o próximo item representará um campo do filtro, ou elemento de filtragem. Literal indica um valor de texto absoluto, ou seja, todos os caracteres são considerados, inclusive espaços em branco. Por exemplo, para definir o campo "dataType" como um campo do filtro, faz-se:

```
<FieldValue>
    <Literal>datatype</Literal>
</FieldValue>
```

Existem dois operadores para especificar o filtro:

- **Equals** – Operador binário que recebe duas strings como operandos. Assume valor true se os operandos são iguais e falso caso contrário. Obs: Esta operação é case insensitive.
- **StartWith** – Operador binário que recebe duas strings como operandos. Verifica se o segundo operando é prefixo do primeiro operando. Obs: Esta operação é case insensitive.

Os elementos de filtragem também podem ser combinados utilizando-se os seguintes operadores:

- **Or**- Operador binário que recebe dois valores booleanos como operandos. Assume valor true se e somente se pelo menos um dos operandos é true
- **And**- Operador binário que recebe dois valores booleanos como operandos. Assume valor true se e somente se ambos os operandos são true.
- **Not**- Operador unário que recebe um valor booleano como operando. Nega o operando. Se ele for true o resultado será false e vice-versa

5 Cache

O framework fornece uma ação listener especial para a implementação de caching. Esta ação é feita pela classe `DefaultCacheListener` do pacote `proxy.actions.listeners`, que apenas ativa ou desativa uma determinada política de cache. O importante é que esta classe recebe como parâmetro o tipo de política de cache que deverá ser implantada. Esse parâmetro, chamado **cacheClassName,** indica uma classe no arquivo de configuração XML. Portanto, a política de cache é uma classe que pode ser definida pelo desenvolvedor de acordo com as necessidades da aplicação. A política de cache deve obedecer à interface definida em `proxy.cache.Cacher`. Esta interface requisita que dois métodos sejam implementados: uma para o armazenamento de uma mensagem e o outro para recuperação de todas as mensagens armazenadas no cache.

Desta forma, é possível definir diferentes políticas de cache para serem aplicadas em contextos diferentes, por exemplo, pode-se adotar um tipo de cachê em desconexões temporárias e outro tipo em desconexões permanentes. Note, porém, que a definição de desconexão temporária ou permanente é específico por aplicação, e deve ser definida na expressão de contexto.

O framework fornece duas implementações de políticas de cache básicas: `FIFOCacher e NoCacher`. A primeira apenas armazena as mensagens em uma ordem FIFO, sem um limite específico de mensagens. A segunda na verdade não faz cache e simplesmente descarta as mensagens, num momento de desconexão por exemplo. Abaixo segue um exemplo de definição de cache no XML na ocorrência de uma desconexões. O cache para desconexões temporárias (por ex. desconexões por menos de 20 segundos) é do tipo FIFO, ou seja, armazena as mensagens e as retorna da mesma ordem de chegada, não descartando nada. E para desconexões permanentes (maiores de 20s) não há cache.

```
<State> <!—Desconexão Temporária  -->
      <Expression>
          <![CDATA[ OnLine = false AND DeltaT < 20000 ]]>
      </Expression>
      <Action class="moca.core.proxy.actions.listeners.DefaultCacheListener">
         <Parameter name="cacheClassName">moca.core.proxy.cache.FIFOCacher
         </Parameter>
      </Action>
  </State>

<State> <!—Desconexão Permanente  -->
      <Expression>
          <![CDATA[ OnLine = false AND DeltaT > 20000 ]]>
      </Expression>
      <Action class="moca.core.proxy.actions.listeners.DefaultCacheListener">
         <Parameter name="cacheClassName">moca.core.proxy.cache.NoCacher
         </Parameter>
      </Action>
</State>
```

6 Adaptação de Imagens

O proxy framework provê algumas classes para adaptação de imagens. Estas classes se encontram no pacote `proxy.actions.adapters`. Abaixo estão listadas as classes, suas funcionalidades e parâmetros.

Uma observação importante é que a definição dos parâmetros da classe distingue letras maiúsculas de minúsculas, portanto o nome do parâmetro (no XML) deve ser idêntico ao definido em cada classe listada a seguir.

Classe ColorToGrayAdapter

Converte uma imagem colorida para preto e branca (escala de cinza), mantendo o tipo de imagem original. Exemplo de configuração no XML está a seguir.

```
<Action class="moca.core.proxy.actions.adapters.ColorToGrayAdapter "/>
```

Classe ConvertToJPEGAdapter

Converte uma imagem qualquer para o formato JPEG, com uma qualidade de compressão pré-definida (no XML).

O parâmetro *compressionQuality* deve ser especificado.

O valor da qualidade de compressão controla a qualidade da imagem e também a taxa de compressão da mesma. Seu valor deve ser entre 0 e 1, por exemplo:

1	Maior qualidade, nenhuma compressão.
0.75	Qualidade alta, média taxa de compressão.
0.50	Qualidade média, boa taxa de compressão.
0.25	Qualidade baixa, alta taxa de compressão;.

Exemplo de configuração no XML.

```
<Action class="moca.core.proxy.actions.adapters.ConvertToJPEGAdapter">
    <Parameter name="compressionQuality">0.4</Parameter>
</Action>
```

Classe CropCenterAdapter

A imagem é recortada gerando-se uma nova imagem que contém a região retangular central de tamanho igual à altura x largura definidos como parâmetros da classe.

Os parâmetros *width* e *height* definem as bordas da imagem recortada. Um exemplo pode ser visto abaixo:

```
<Action class="moca.core.proxy.actions.adapters.CropCenterAdapter">
    <Parameter name="width">60</Parameter>
    <Parameter name="height">50</Parameter>
</Action>
```

Classe ScaleImageAdapter

Escala a imagem por um fator pré-definido. Se o fator de escala for maior que 1 a imagem é ampliada, por outro lado, se o fator for entre 0 e 1 a imagem é reduzida. Note que se o valor do fator for negativo ele será convertido para 0.01.

O parâmetro *factor* indica o fator de escala que deve ser definido como parâmetro da classe no XML, como no exemplo:

```
<Action class="moca.core.proxy.actions.adapters.ScaleImageAdapter">
    <Parameter name="factor">0.5</Parameter>
</Action>
```

7 Manutenção dos clientes atendidos pelo proxy

O ProxyFramework mantém uma lista de clientes para os quais ele intermedeia a comunicação e executa adaptações se necessário. Entretanto ele apenas intermedeia a comunicação de clientes que estejam cadastrados no mesmo. A gerência de clientes fica, então, a cargo da aplicação, para uma maior flexibilidade e possibilitar que, por exemplo, a aplicação Proxy autentique o usuário, e só depois passe a aceitar suas mensagens. Outro exemplo seria poder remover um usuário não apenas baseado em dados técnicos(físicos – como estar desconectado a um certo tempo) mas também por questões de negócio, por exemplo, o cliente não pagou pelo serviço.

Desta forma, para a correta manutenção da lista, cada cliente deve ser explicitamente cadastrado no Proxy, e removido quando desejado. Para isso o desenvolvedor deve usar os métodos `addClient e removeClient` da classe ProxyFramework. Além disso, é necessário registrar no serviço CIS da MoCA aqueles clientes que a aplicação deseje que o contexto seja monitorado. Para este propósito deve-se usar o método `registerClientAtCIS`, ou também já setar esta opção no momento da inserção do cliente `(addClient)`. Para a remoção de clientes apenas do serviço CIS pode-se usar `removeClientFromCIS` ou `removeAllClientsFromCIS`.

Uma observação é que remover clientes do CIS não os remove da lista do Proxy e, portanto, este continua aceitando suas mensagens. Mas a remoção de um cliente do proxy implica na automática remoção do mesmo no MoCA CIS.

Ao receber a mensagem de um cliente não cadastrado, o proxyFramework a repassa para a aplicação proxy para que esta decida se o cliente deve ou não ser cadastrado no framework. Apesar de receber mensagens de qualquer cliente, se este não for cadastrado, o proxy não terá como responder (i.e. enviar mensagens) a ele.

8 Conectividade de clientes

O Proxy Framework oferece a opção da aplicação proxy ter conhecimento sobre a conectividade de clientes. Isso pode ser usado, por exemplo, para calcular por quanto tempo os clientes estão desconectados, e então decidir se um cliente deve ser removido do proxy caso exceda um determinado tempo de desconexão.

A aplicação pode receber notificações quando clientes se conectam e se desconectam. Na notificação, é informado a identificação do cliente e seu estado, conectado ou não. Para que a aplicação proxy seja notificada, o cliente deve ter sido cadastrado no CIS, como explicado na seção anterior. Além disso, a aplicação deve implementar a interface `ConnectionListener` encontrada no pacote `proxy.development` (método `onReceiveConnectionNotification`), e cadastrar este listener no framework através do método `addClientConnectedListener` da classe `proxy.ProxyFramwork`.

9 Liberação de recursos

È importante salientar que o desenvolvedor deve invocar o método `freeResources` da classe `ProxyFramework` ao final de sua aplicação para uma correta liberação dos recursos alocados. Isto serve para, dentre outras coisas, finalizar iterações com serviços da MoCA.

10 Arquivos de configuração

O diretório `config` contém os arquivos de configuração externa, ou seja, aqueles arquivos passíveis de modificação por usuários do framework. Um exemplo deste tipo de arquivo é o ProxyConf.xml (exemplificado na seção 3), que estabelece diversas regras de adaptação para o Proxy.

Outros arquivos importantes, dizem respeito à configuração do proxy e de serviços da MoCA. O arquivo proxy.properties contém as informações sobre endereço IP e portas para do proxy atuando como servidor para clientes móveis, e o endereço público (IP) do proxy na sua interface com o servidor da aplicação (papél de cliente). Abaixo está um exemplo deste arquivo, com as tags que devem ser definidas.

```
proxy.server.ip_address=0.0.0.0
proxy.sync_server.port=55100
proxy.event_server.port=55110
proxy.public_local.ip_address=139.82.24.232
```

A configuração dos endereços dos serviços MoCA usados pelo proxy são definidos no arquivo moca.properties. A seguir há um exemplo deste arquivo. No momento os serviços CIS e DS são usados, e devem ser definidos de acordo com as tags definidas abaixo:

```
cis.server.host=cis.lac.inf.puc-rio.br
cis.server.port=55001
cis.publisher.port=55000
ds.host=ds.lac.inf.puc-rio.br
```

11 Distribuição

Em conjunto com o código fonte, é disponibilizado um script ant para compilação, geração de documentação (javadoc), execução de teste automático (Junit) e criação do diretório de distribuição.

A ultima tarefa, executada pelo `target dist`, cria um diretório homônimo com todos arquivos necessários em uma distribuição binária do framework. Na Figura 3 é mostrada a estrutura de diretórios abaixo de dist.:

Figura 3: Estrutura de Diretórios de Distribuição

Como pode ser visto na figura acima, o diretório de distribuição é composto basicamente por 3 subdiretórios:

- config – contém os arquivos de configuração externa, aqueles arquivos passiveis de modificação por usuários da distribuição binária do framework. Um exemplo deste tipo de arquivo é o ProxyConf.xml, que estabelece diversas regras de adaptação para o Proxy.
- doc – documentação JavaDoc da API do framework..
- `lib` – contém todos os arquivos jars necessários para a execução do framework. Os jars do framework estarão logo abaixo do diretório lib. As demais dependências (dependências externas) estarão subdivididas em diretórios cujos nomes descrevem sua procedência. É importante lembrar que estas dependências (ou versões compatíveis destas) deverão constar no classpath de qualquer aplicação baseada sobre o framework.